

Inheritance & Polymorphism

Inheritance & Polymorphism

- Inheritance

Classes (subclasses) are created from existing ones (superclasses).

An inheritance relationship means that the subclass inherits members (instance variables and methods) of the superclass.

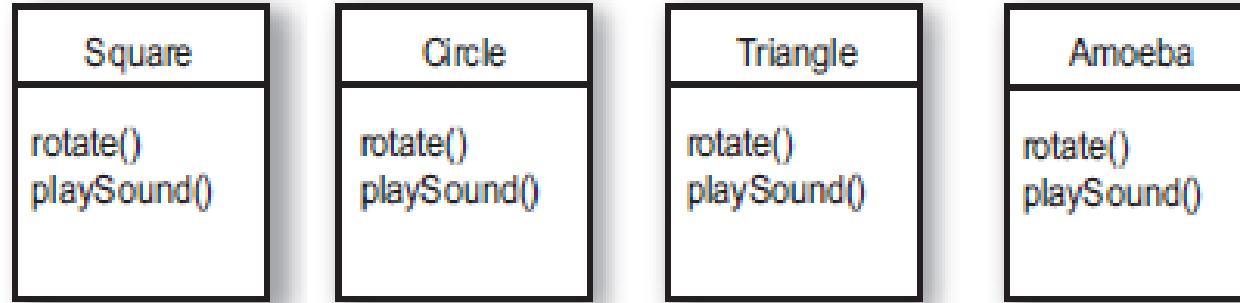
- Facilitate software reusability
 - Absorbing attributes and behaviors.
 - Adding new capabilities (Subclass can add new instance variables and methods).
 - Subclass usually adds instance variables and methods.
- Inheritance is used to avoid duplicate codes

- Polymorphism

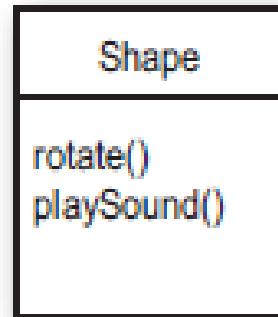
- Enables developers to write programs in general fashion
 - Handle variety of existing and yet-to-be-specified classes
- Helps add new capabilities to system

Inheritance

- Consider the following classes:

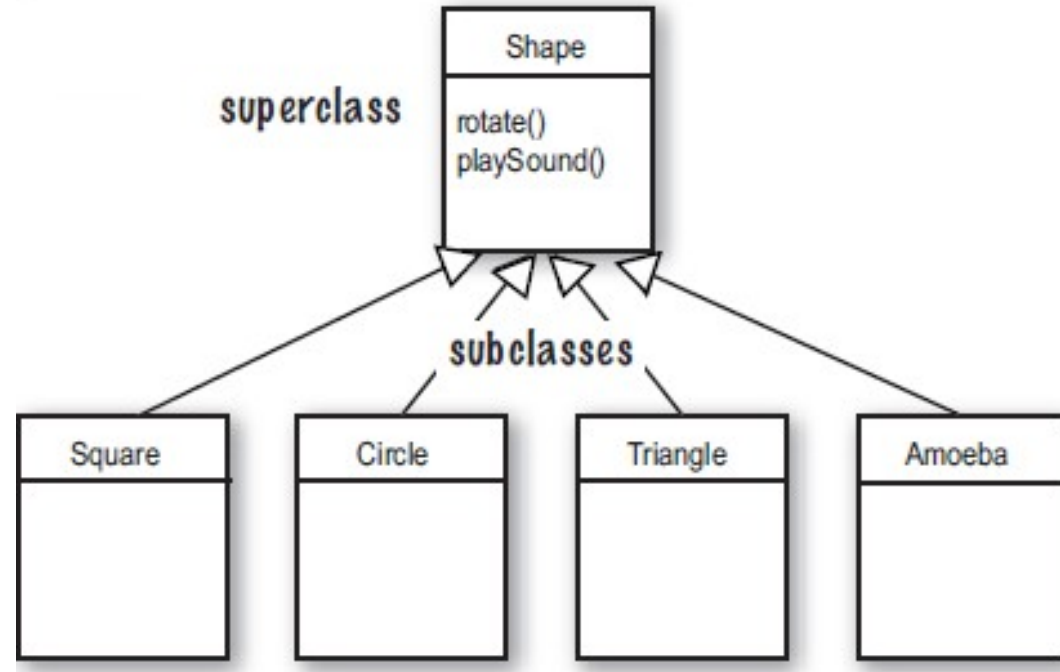


- Look at what all four classes have in common.
 - They are all shapes, they can rotate and play sound
- Extract out the common features, put them in a new class called Shape



Inheritance

- Then link the other four shape classes to the new Shape class, in a relationship called inheritance.

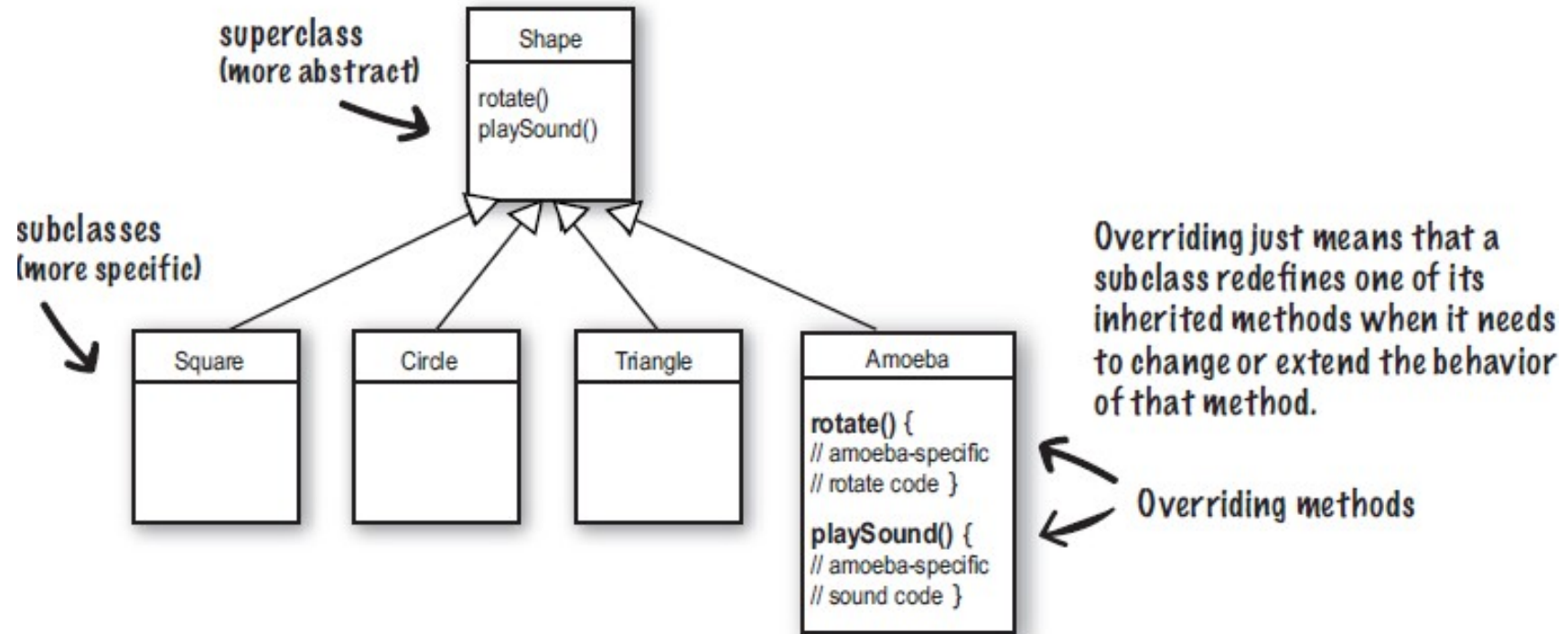


- Square **inherit** from Shape (Square **extends** Shape),
- Circle **inherit** from Shape (Circle **extends** Shape) etc.
- The methods `rotate()` and `playsound()` are removed from all four shapes, now there is **only one copy in the superclass to maintain**

Inheritance

Challenge: Amoeba might have completely different ways of rotating and different sound.

Trick: The amoeba class can override the methods of the Shape class. At runtime, the JVM knows exactly which rotate() method to run when someone tells the amoeba to rotate.



Instance variables are **not overridden** because they don't define any special behavior, so a subclass can give an inherited instance variable any value it chooses.

Inheritance (Designing an Inheritance Tree of Animals)



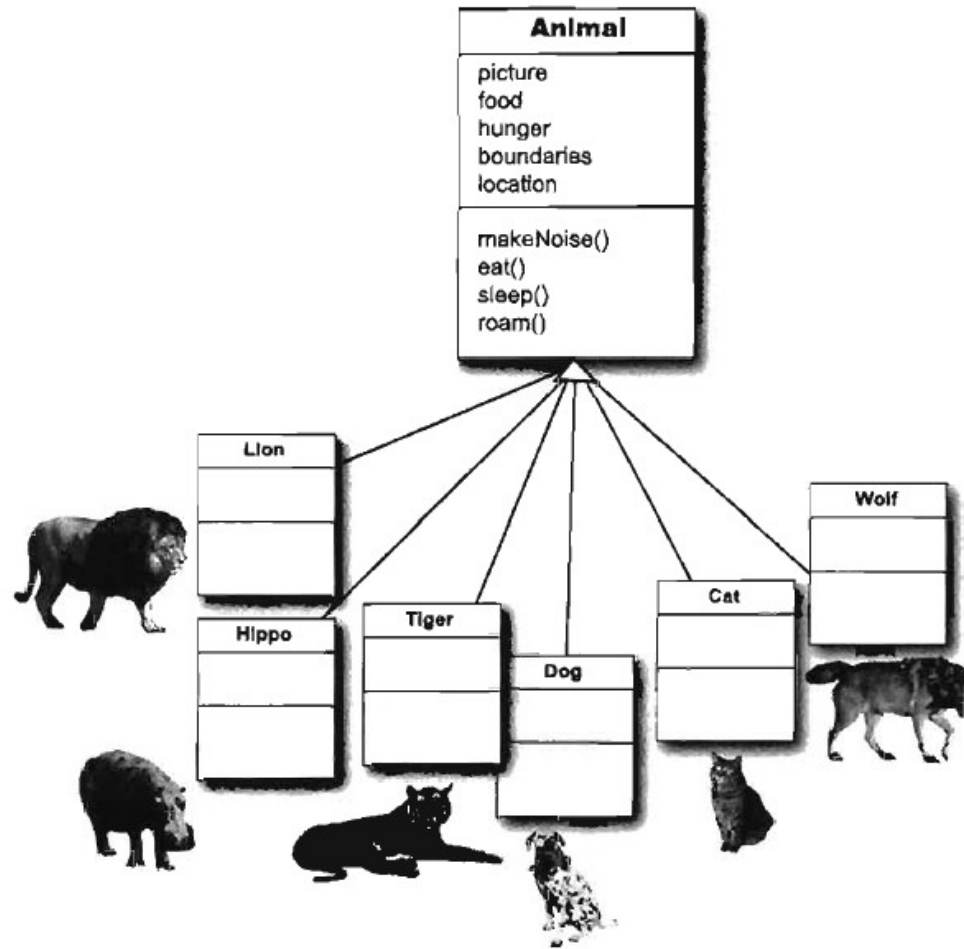
What do these six types have in common? This helps you to abstract out behaviors.

How are these types related? This helps you to define the inheritance tree relationships



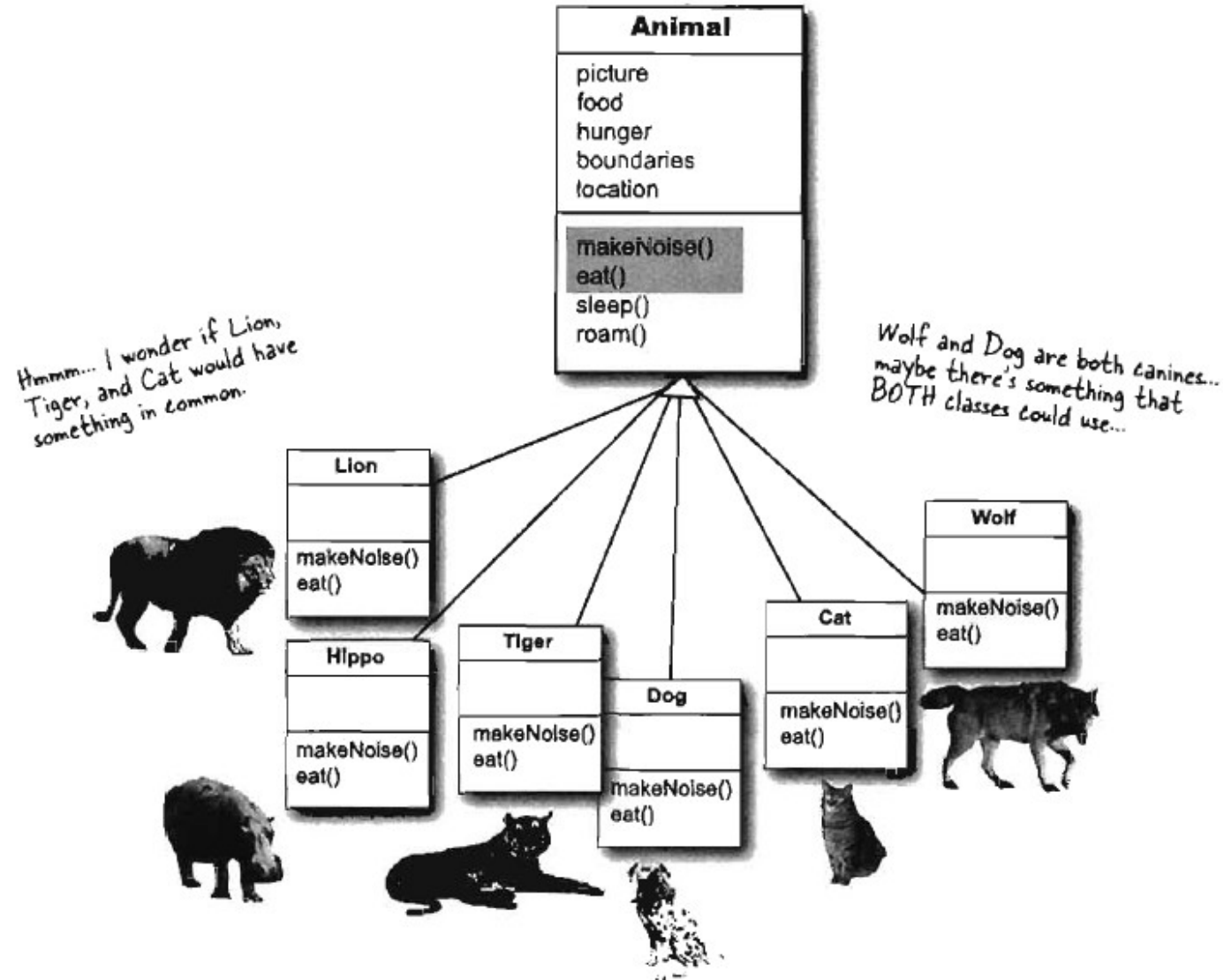
- These objects are **all animals**, so make a common **superclass** called **Animal**
- Put in instance variables and methods that all objects (animals) might need

Inheritance (Designing an Inheritance Tree of Animals)



Do all animals eat the same way? Which methods can be overridden?

Inheritance (Designing an Inheritance Tree of Animals)

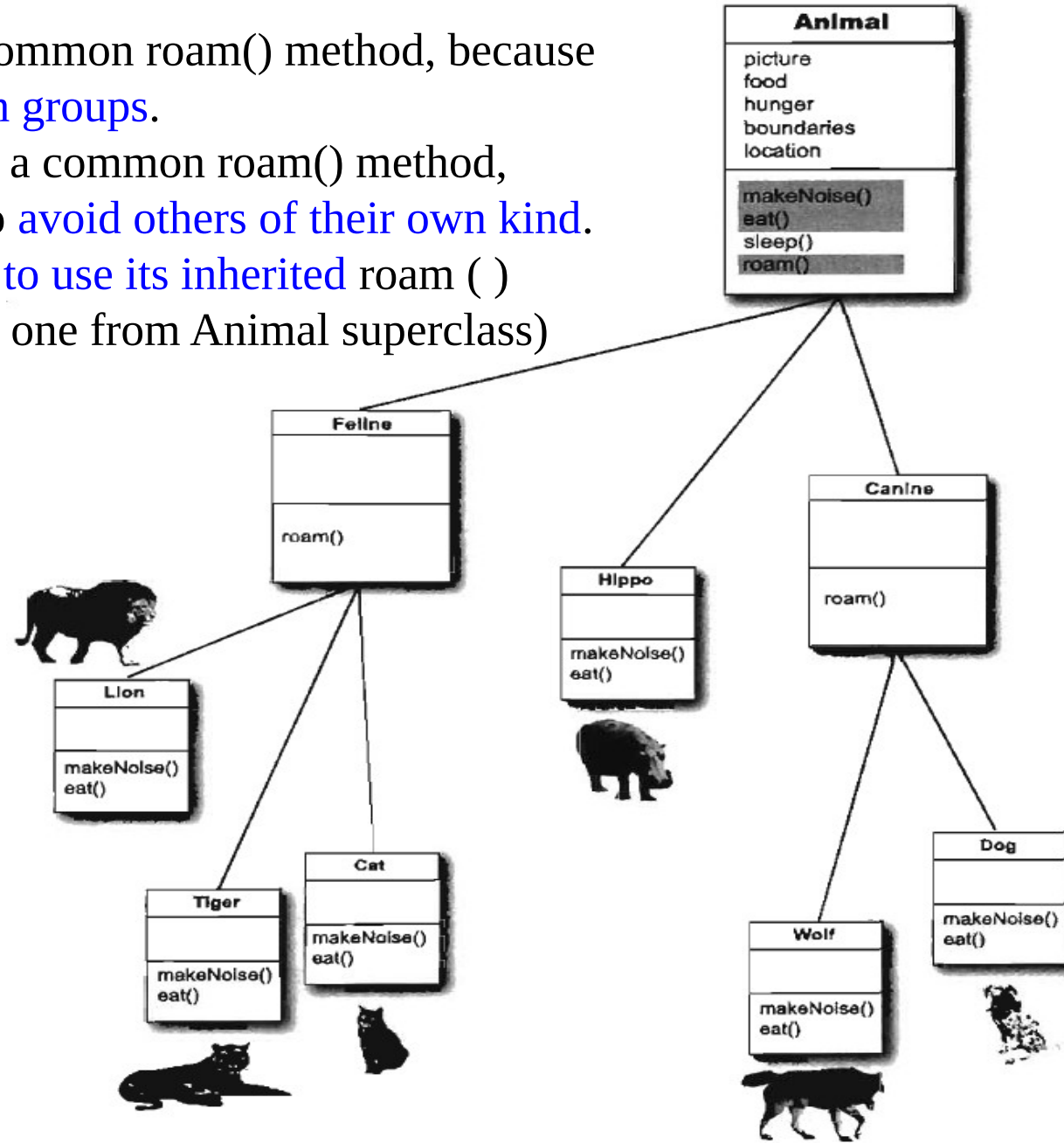


Inheritance (Designing an Inheritance Tree of Animals)

Canines can use a common roam() method, because they tend to **move in groups**.

Felines can also use a common roam() method, because they tend to **avoid others of their own kind**.

Hippo can **continue to use its inherited** roam () method (the generic one from Animal superclass)



Inheritance (Designing an Inheritance Tree of Animals)

1. How many methods does the wolf class has?
2. In the following statements, which method is called in the inheritance tree?

make a new Wolf object

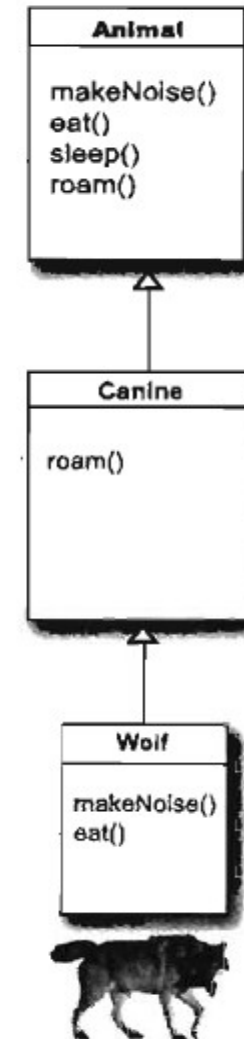
```
Wolf w = new Wolf();
```

```
w.makeNoise();
```

```
w.roam();
```

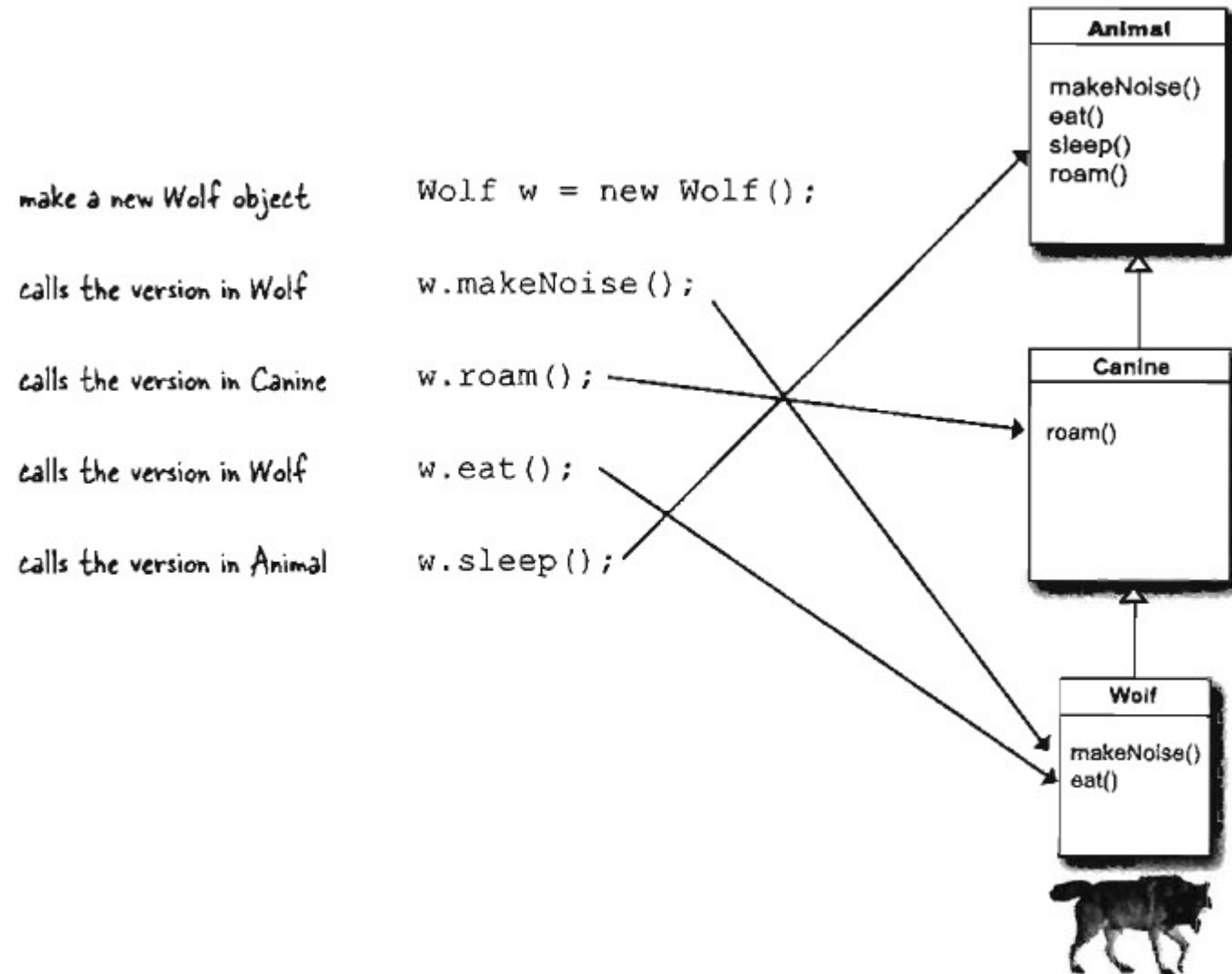
```
w.eat();
```

```
w.sleep();
```



Inheritance (Designing an Inheritance Tree of Animals)

How many methods does the wolf class has? (Four 4)



Inheritance (IS-A Test)

Remember that when one class inherits from another, we say that the subclass extends the superclass. IS-A test is used in order to know if one thing can extend another.

Examples:

Triangle IS-A Shape, that's true.

Cat IS-A Feline, that works too .

Surgeon IS-A Doctor, still good.

Note: The inheritance IS-A relationship works in only one direction.

Canine extends Animal

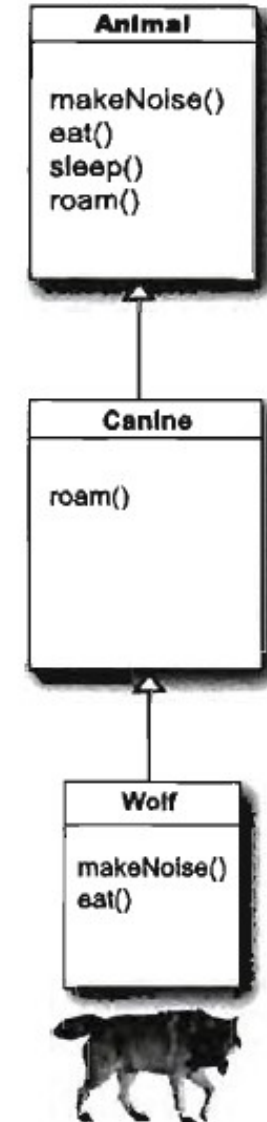
Wolf extends Canine

Wolf extends Animal

Canine IS-A Animal

Wolf IS-A Canine

Wolf IS-A Animal



Superclass & Subclass Members Accessibility

- A superclass's **public members are accessible anywhere** the program has a reference to that superclass type or one of its subclass types.
- A superclass's **private members** are accessible only in methods of that superclass.
- A superclass's **protected access members serve as an intermediate level of protection** between **public and private access**.
- **A superclass's protected members** may be accessed only by methods of the superclass, by methods of subclasses and by methods of other classes in the same package (**protected members** have package access).
- When a subclass method overrides a superclass method, the superclass method may be accessed from the subclass by preceding the superclass method name with keyword **super followed by the dot operator (.)**.

Accessing Superclass Members

If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword `super`. Consider example:

```
public class Superclass {
```

```
    public void printMethod() {
```

```
        System.out.println("Printed in Superclass.");
```

```
    }
```

```
}
```

//Here is a subclass that overrides printMethod():

```
public class Subclass extends Superclass {
```

```
    // overrides printMethod in Superclass
```

```
    public void printMethod() {
```

```
        super.printMethod();
```

```
        System.out.println("Additional info in Subclass");
```

```
    }
```

```
    public static void main(String[] args) {
```

```
        Subclass s = new Subclass();
```

```
        s.printMethod();
```

```
    }
```

```
}
```

Within Subclass, the method `printMethod()` refers to the one declared in Subclass, which overrides the one in Superclass. So, to refer to `printMethod()` inherited from Superclass, Subclass must use a qualified name, using `super` as shown.

Compiling and executing Subclass prints the following:

```
Printed in Superclass.
```

```
Additional info in Subclass
```

Overriding & Overloading Methods

When you **override** a method from a superclass, you need to keep in mind the following rules:

1.Arguments must be the same, and return types must be compatible.

- The Superclass defines how other code can use a method.
- Whatever the superclass takes as an argument. the subclass overriding the method must use that same argument.
- And whatever the superclass declares as a return type, the overriding method must declare either the same type or a subclass type .

2. The method can't be less accessible.

- That means the access level must be the same. That means you can't, for example, override a public method and make it private.

Overriding & Overloading Methods

- **Method overloading** is nothing more than having two or more methods with the same name but different argument lists.
- Overloading lets you make **multiple versions of a method, with different argument lists**, for convenience to the callers. It is useful in constructors.
- An overloaded method is just a different method that happens to have the same method name, It has **nothing to do with inheritance and polymorphism**. An overloaded method is NOT the same as an overridden method.

Rules:

1. The return types can be different.
2. You can't change ONLY the return type.
 - To overload a method, you MUST change the argument list, although you *can* change the return type to anything. If only the return type is different, *the compiler will assume you're trying to wrongly override the method*.
3. You can vary the access levels in any direction.

Overriding & Overloading Methods

Legal examples of method overloading:

```
public class Overloads {  
  
    String uniqueID;  
  
    public int addNums(int a, int b) {  
        return a + b;  
    }  
  
    public double addNums(double a, double b) {  
        return a + b;  
    }  
  
    public void setUniqueID(String theID) {  
        // lots of validation code, and then:  
        uniqueID = theID;  
    }  
  
    public void setUniqueID(int ssNumber) {  
        String numString = "" + ssNumber;  
        setUniqueID(numString);  
    }  
}
```

Final Methods and Classes

It is also possible to define methods and classes with the **final modifier**.

- A **method** that is declared **final** cannot be **overridden** in a subclass.
- **Methods that are** declared **static and** methods that are declared **private** are implicitly **final**.

A class that is declared **final** cannot be a **superclass** (i.e., No any other class can inherit from a final class). **All methods in a final class are implicitly final.**

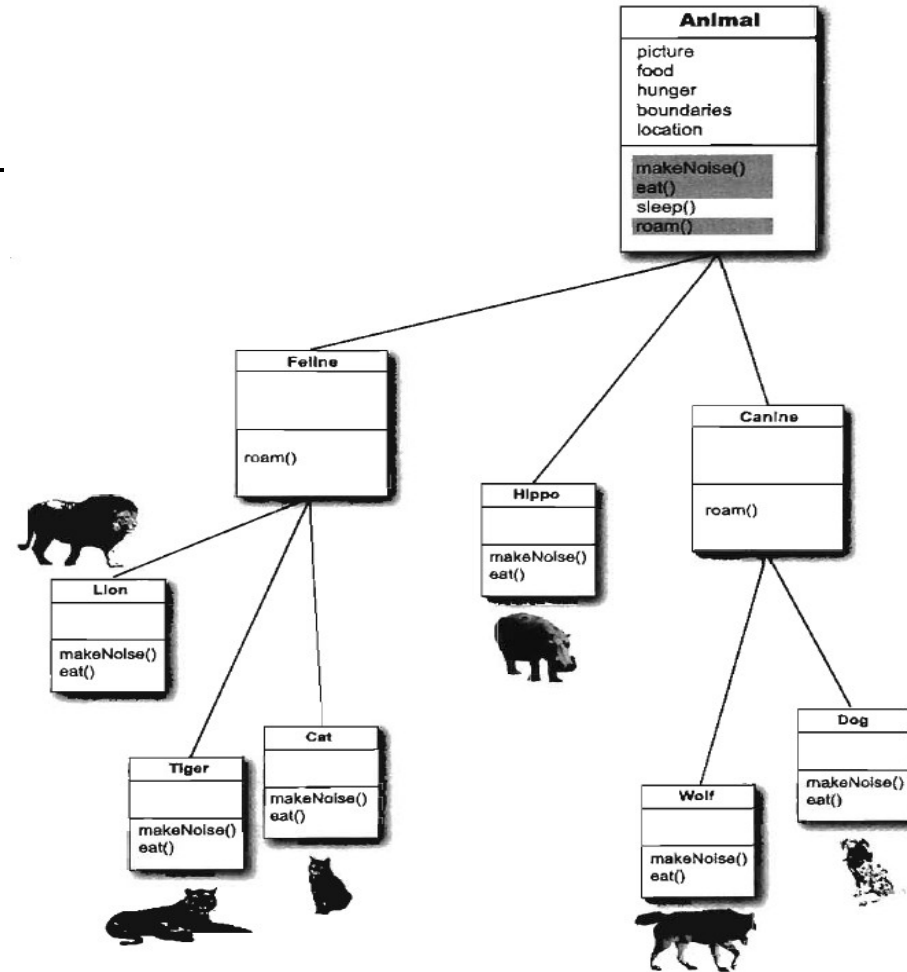
Typically, you won't make your classes final, unless for security reasons or to make sure that the methods will always work the way they were written (because they can't be overridden).

A lot of classes in the Java API are final for that reason.

Abstract class

Designed so that duplicate code is kept to a minimum

Overriden methods that need subclass-specific implementation

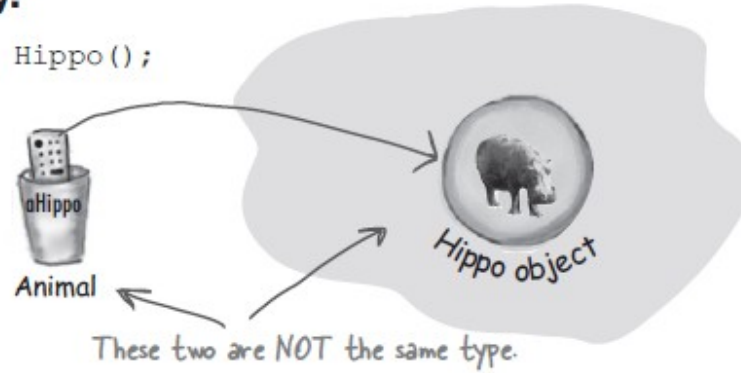


Abstract class

And we know we can say:

```
Animal aHippo = new Hippo();
```

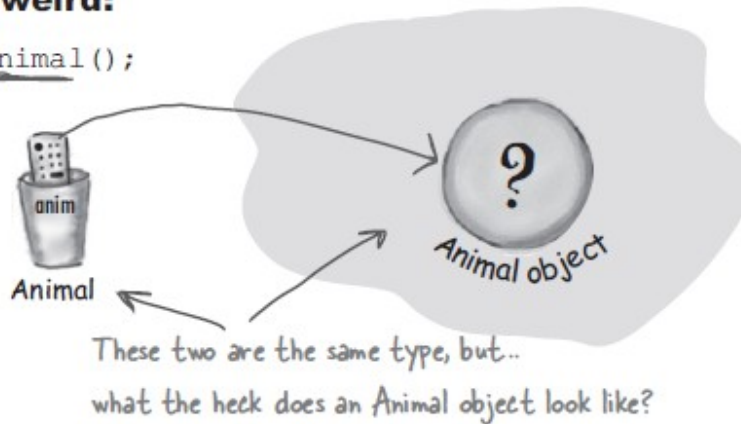
Animal reference to
a Hippo object.



But here's where it gets weird:

```
Animal anim = new Animal();
```

Animal reference to
an Animal object.



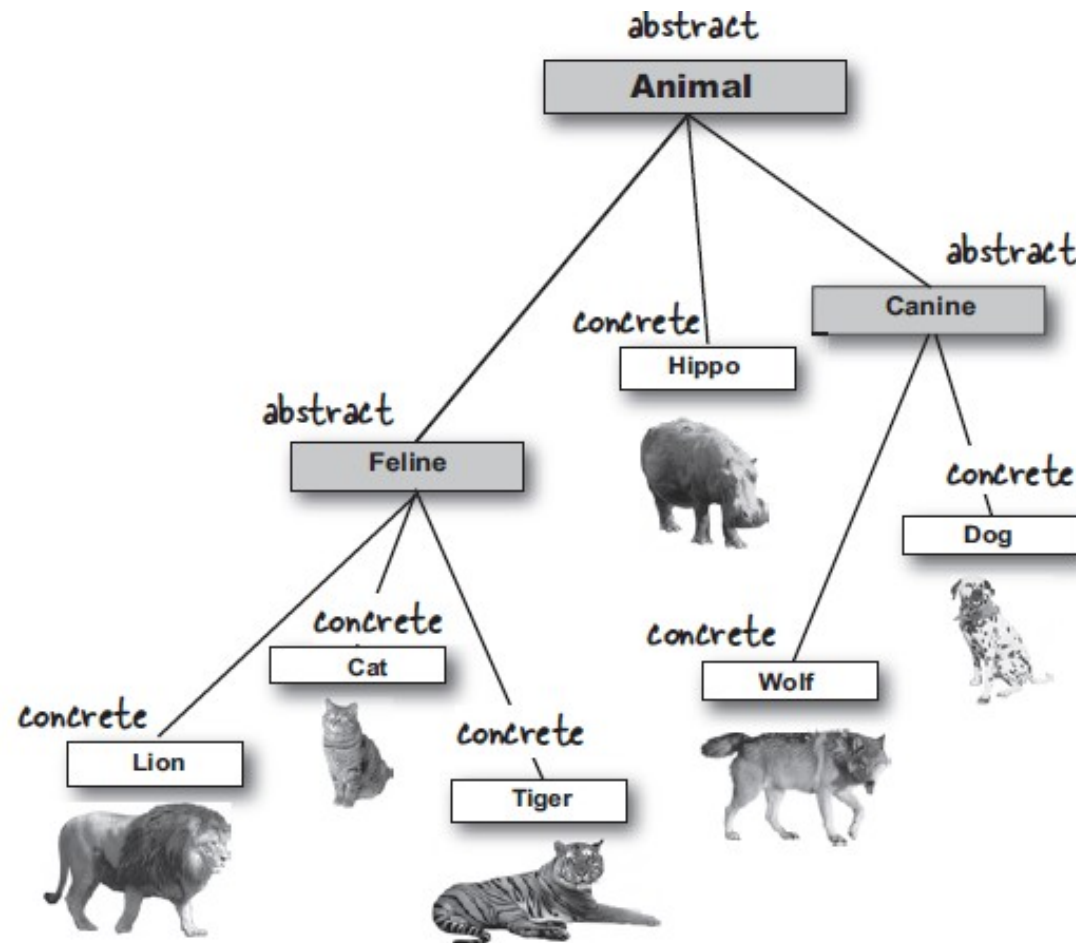
It makes sense to create a hippo object or a tiger object, but **what exactly is an animal object?** What shape is it? Color? # of legs?

Abstract class

- Animal class is needed for inheritance and polymorphism.
- But we want programmers to instantiate only subclasses, not Animal superclass itself.
- To prevent a class from ever being instantiated, is done by making the class an abstract class, the compiler will stop anycode from ever creating an instance of that type.
- An abstract class is the type of a class that can not be instantiated.

Abstract vs concrete classes

• An abstract class has virtually no use, no value, no purpose in life, unless it is extended. A class that is not abstract is called a **concrete class**. In the Animal Inheritance tree, we can declare Animal, canine and feline classes abstract & hence Hippo, Tiger, Wolf, Dog, Lion and Cat become concrete classes.



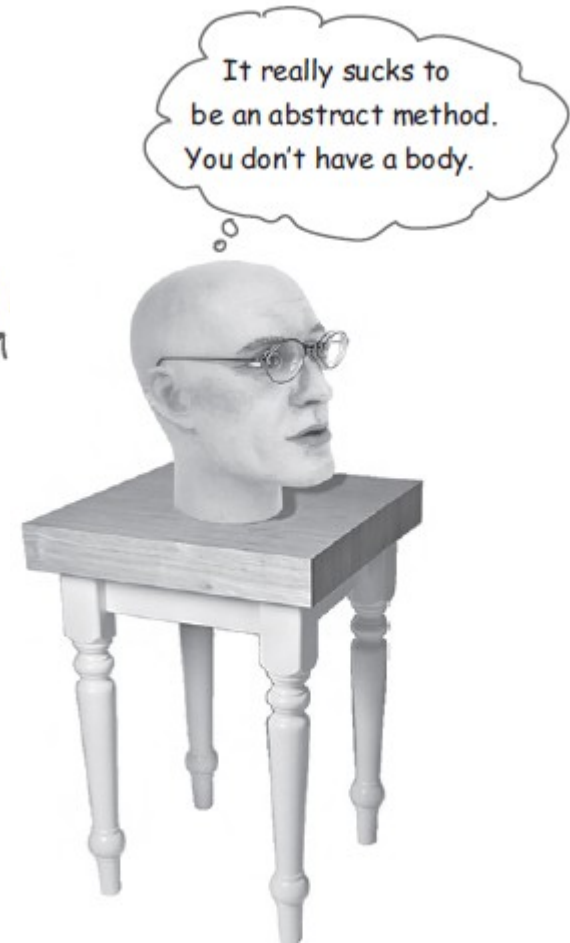
Abstract Methods

- Besides classes, you can mark methods abstract too.
- An abstract class means the class must be extended, an **abstract method must be overridden**.
- An **abstract method has no body**.

```
public abstract void eat()
```

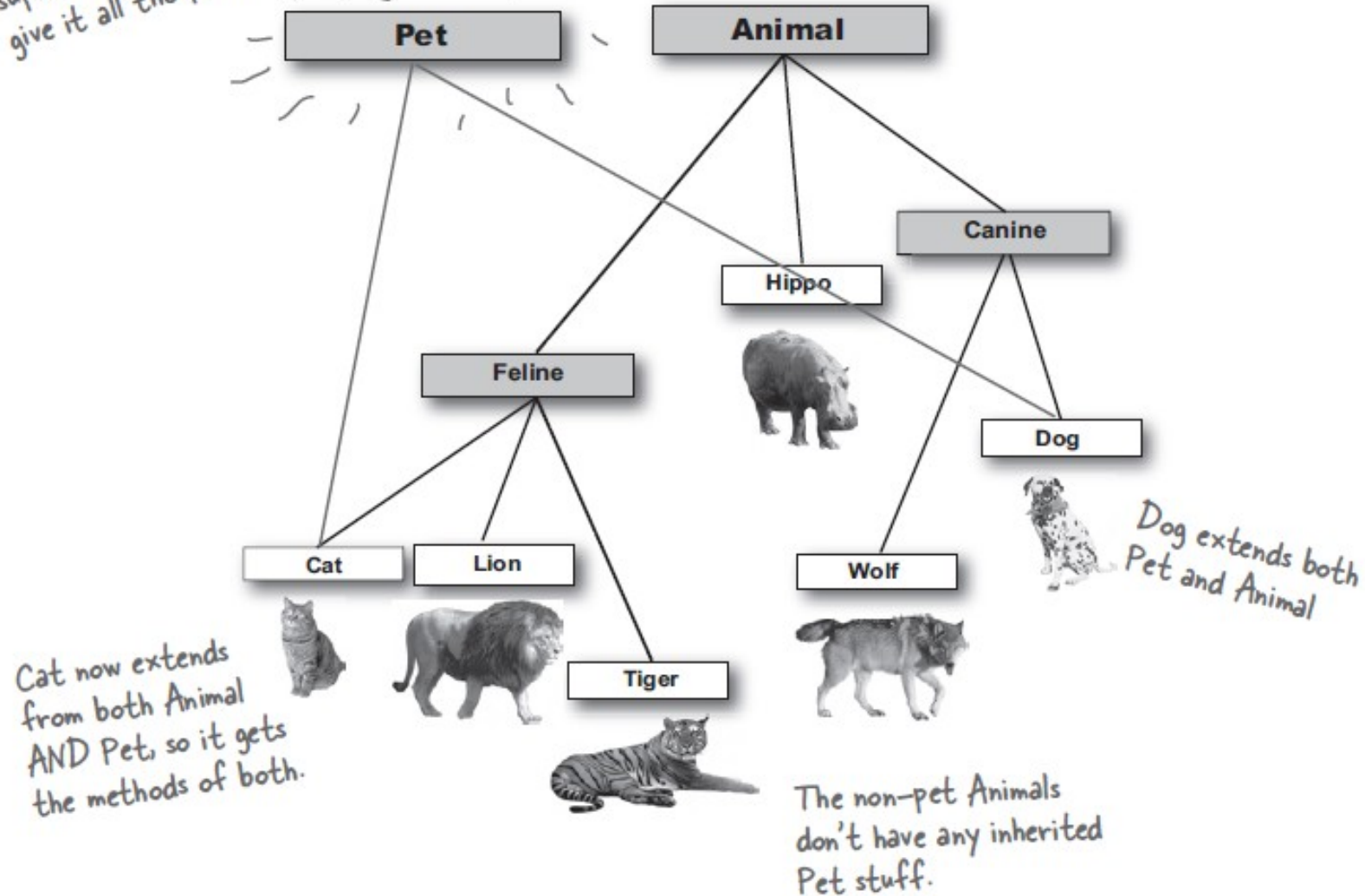
No method body!
End it with a semicolon.

- If a **method is declared abstract**, then the **class must be marked abstract as well**. There is no abstract method in a non-abstract class.



Multiple Inheritance

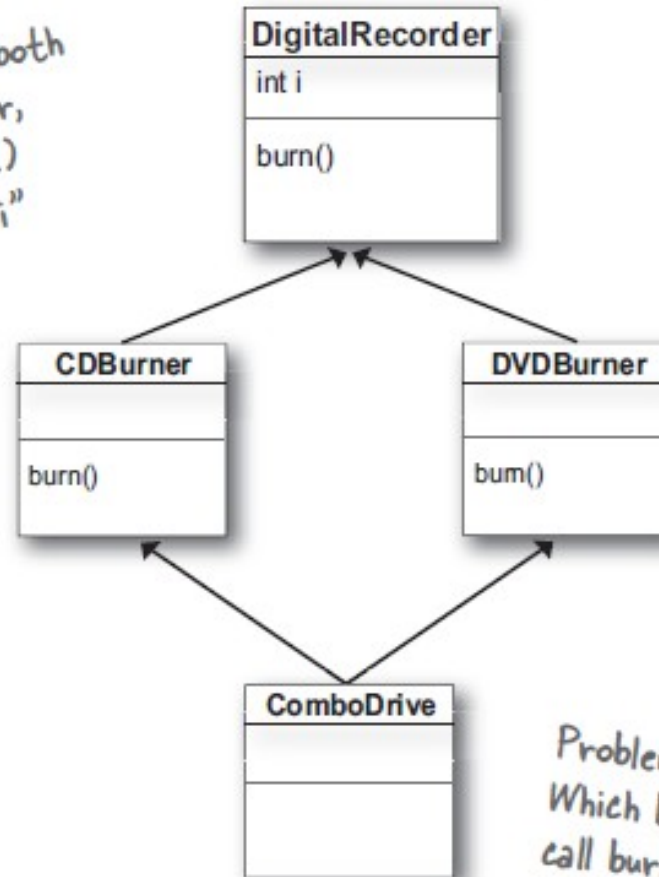
We make a new abstract superclass called Pet, and give it all the pet methods.



Deadly Diamond of Death-DDD

Deadly Diamond of Death

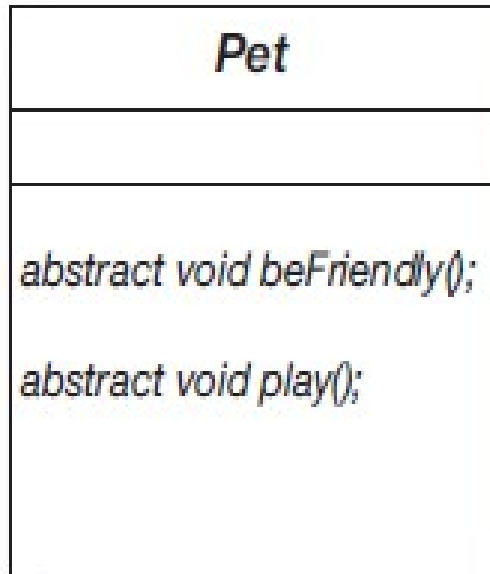
CDBurner and DVDBurner both inherit from DigitalRecorder, and both override the burn() method. Both inherit the "i" instance variable.



Imagine that the "i" instance variable is used by both CDBurner and DVDBurner, with different values. What happens if ComboDrive needs to use both values of "i"?

Problem with multiple inheritance. Which burn() method runs when you call burn() on the ComboDrive?

DDD-> Interface to the Rescue!



**A Java interface is like a
100% pure abstract class.**

*All methods in an interface are
abstract, so any class that IS-A
Pet MUST implement (i.e. override)
the methods of Pet.*

- Prior to the release of Java 8 an interface could contain only abstract methods—but now an interface can also contain static methods and default methods.
- A default method is a new concept specifically to be used in interfaces, if required. A default method is a regular method with a complete implementation and so is automatically inherited by all classes that implement the interface (though the implementing class may override this implementation if it chooses).
- Adding default methods into interfaces means that additions can be made to an interface without every class that implements the previous version of that interface having to change (as would be the case if we added new abstract methods to an interface).

Making and implementing Interface

Making and Implementing the Pet interface

You say 'interface' instead of 'class' here

```
public interface Pet {  
    public abstract void beFriendly();  
    public abstract void play();  
}
```

interface methods are implicitly public and abstract, so typing in 'public' and 'abstract' is optional (in fact, it's not considered 'good style' to type the words in, but we did here just to reinforce it, and because we've never been slaves to fashion...)

All interface methods are abstract, so they **MUST** end in semicolons. Remember, they have no body!

Dog IS-A Animal
and Dog IS-A Pet

```
public class Dog extends Canine implements Pet {
```

You say 'implements' followed by the name of the interface.

```
    public void beFriendly() {...}
```

```
    public void play() {...}
```

```
    public void roam() {...}
```

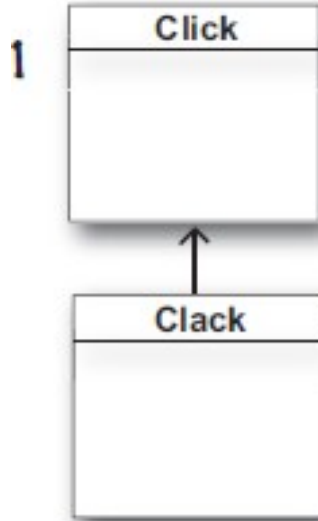
```
    public void eat() {...}
```

You SAID you are a Pet, so you **MUST** implement the Pet methods. It's your contract. Notice the curly braces instead of semicolons.

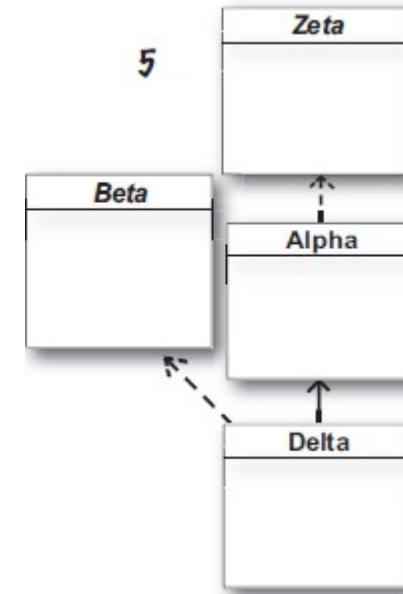
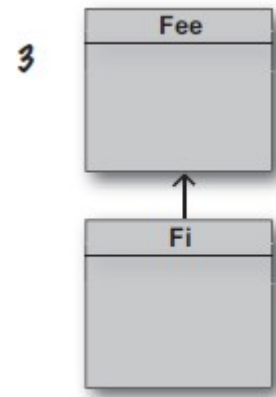
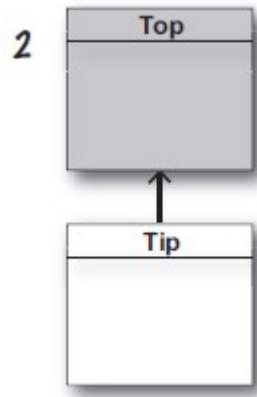
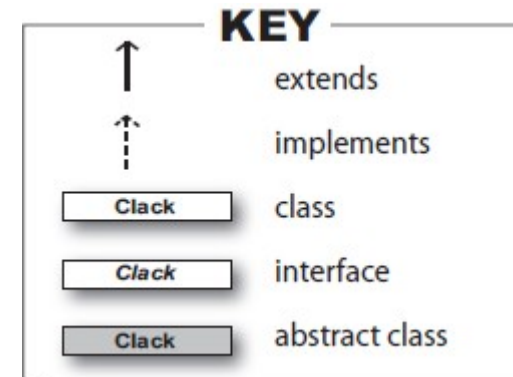
These are just normal overriding methods.

```
}
```

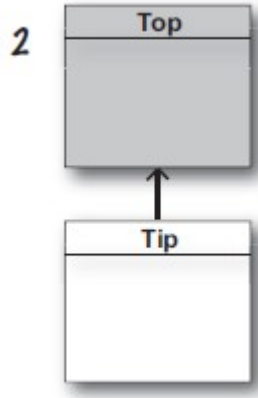
Write the valid java declaration



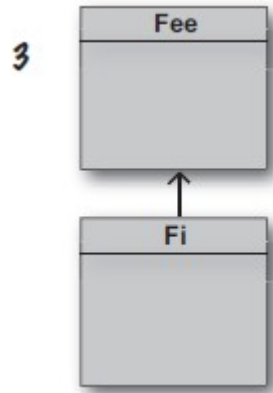
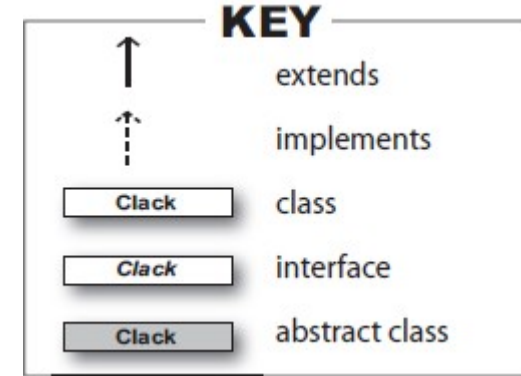
1) `public class Click { }`
`public class Clack extends Click { }`



The valid java declaration

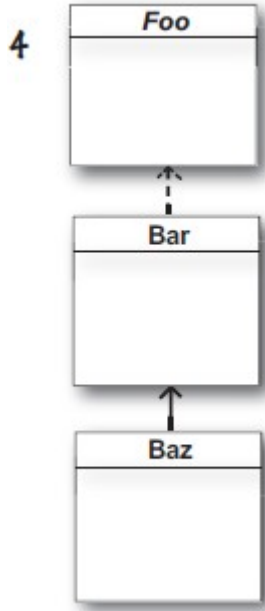


2) `public abstract class Top { }`
`public class Tip extends Top { }`

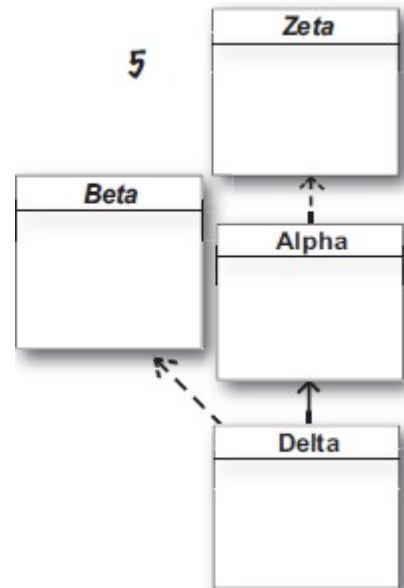
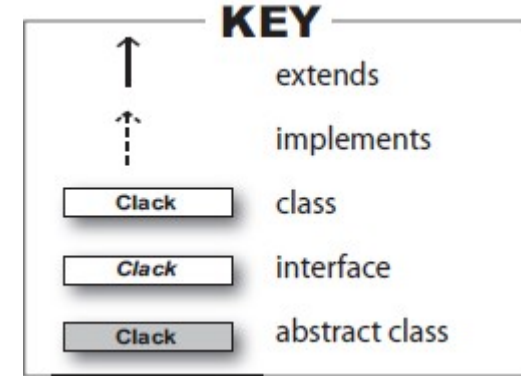


3) `public abstract class Fee { }`
`public abstract class Fi extends Fee { }`

The valid java declaration



4) `public interface Foo { }`
`public class Bar implements Foo { }`
`public class Baz extends Bar { }`



5) `public interface Zeta { }`
`public class Alpha implements Zeta { }`
`public interface Beta { }`
`public class Delta extends Alpha implements Beta { }`

Polymorphism

Polymorphism literally means having many forms, and it is an important feature of object-oriented programming languages. It refers, in general, to the phenomenon of having methods and operators with the same name performing different functions.

Different ways in which polymorphism can be achieved:

1. Operator overloading
2. Method overloading
3. Method overriding
4. Type polymorphism

Polymorphism

Different ways in which polymorphism can be achieved:

Operator overloading

Operators that are overloaded behave differently depending on the type of data they are manipulating.

The + operator, for example, can be used for the concatenation of strings as well as for addition. It should be noted that Java, as opposed to some other languages, does not allow the user to overload operators.

Method overloading

Several methods in a class have the same name and are distinguished by their parameter lists. It is particularly useful for defining a number of different constructors.

Method overriding

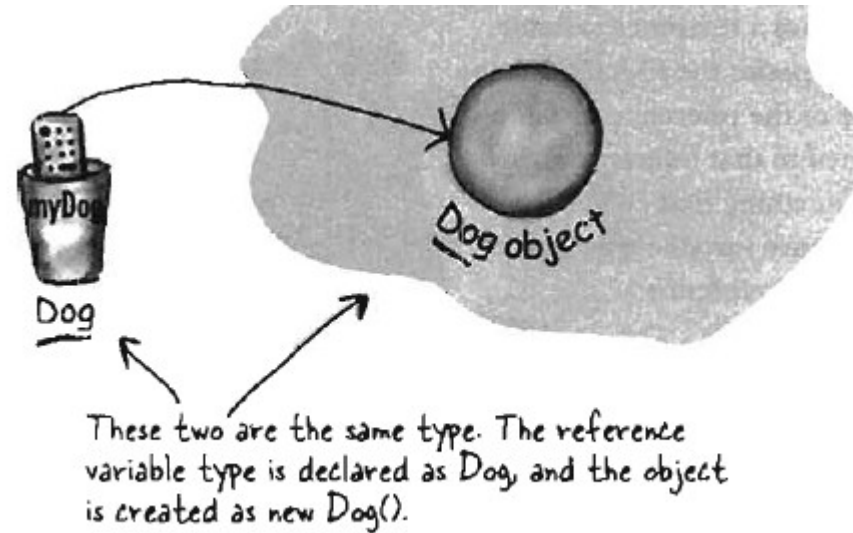
Method overriding is a way of achieving polymorphism by redefining a method of a class within a subclass. Here methods are distinguished not by their parameter lists but by the object with which they are associated. Method overriding is made powerful by having the ability to define abstract methods and therefore guaranteeing the existence of these methods further down the hierarchy.

Type polymorphism

Type polymorphism refers to the technique by which values of different types can be handled in a uniform way. Examples of this are the `System.out.print` and the `System.out.println` methods which are set up to accept objects of many different types—`int`, `char`, `double`, `String` and so on. This is achieved, of course, by defining many different (overloaded) methods. In this way it is possible for a single method to appear to accept an object of any type. This is known as parametric polymorphism.

Polymorphism

- Polymorphism means many forms:
 - Enables developers to write programs in general fashion
 - Handle variety of existing and yet-to-be-specified classes
 - Helps add new capabilities to system
- Recall the variable declaration and assignment

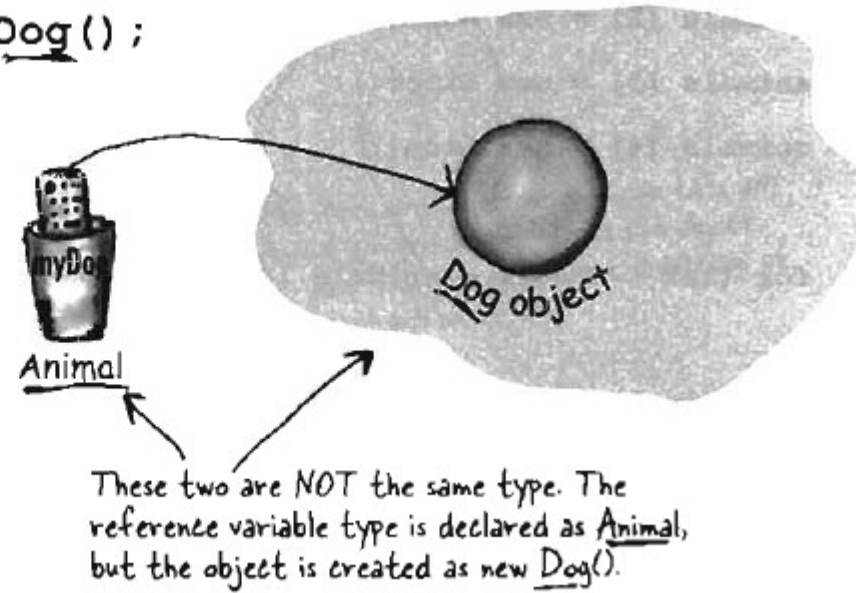


- The important point is that the reference type AND the object type are the same.

Polymorphism

- But with polymorphism, the reference and the object can be different.

```
Animal myDog = new Dog();
```



With polymorphism, the reference type can be a superclass of the actual object type.

Polymorphism

With polymorphism, when a reference variable is declared, any object that passes the IS-A test for the declared type of the reference variable can be assigned to that reference.

i.e anything that *extends the* declared reference variable type can be *assigned to the* reference variable. *Therefore it is possible to create a polymorphic arrays.*

```
Animal[] animals = new Animal[5];
```

Declare an array of type Animal. In other words, an array that will hold objects of type Animal.

```
animals [0] = new Dog();
```

```
animals [1] = new Cat();
```

```
animals [2] = new Wolf();
```

```
animals [3] = new Hippo();
```

```
animals [4] = new Lion();
```

But look what you get to do... you can put ANY subclass of Animal in the Animal array!

```
for (int i = 0; i < animals.length; i++) {
```

And here's the best polymorphic part (the *raison d'être* for the whole example), you get to loop through the array and call one of the Animal-class methods, and every object does the right thing!

```
    animals[i].eat();
```

```
    animals[i].roam();
```

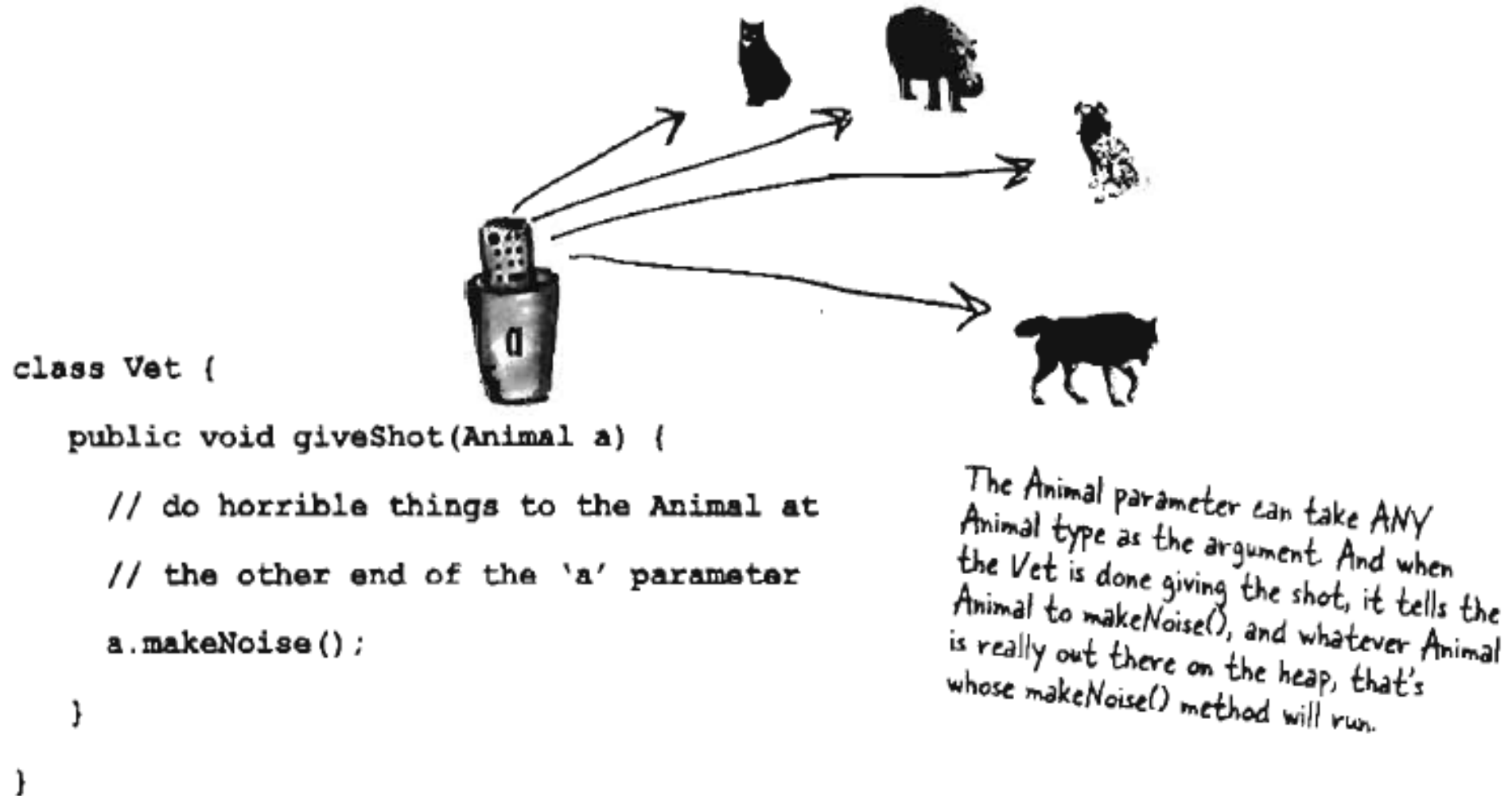
When 'i' is 0, a Dog is at index 0 in the array, so you get the Dog's eat() method. When 'i' is 1, you get the Cat's eat() method

Same with roam().

```
}
```

Polymorphism

- It is also possible to have polymorphic arguments and return types.



Polymorphism

- It is also possible to have polymorphic arguments and return types.

```
class PetOwner {
```

```
    public void start() {
```

```
        Vet v = new Vet();
```

```
        Dog d = new Dog();
```

```
        Hippo h = new Hippo();
```


```
        v.giveShot(d);
```

```
        v.giveShot(h);
```

```
    }
```

```
}
```

The Vet's giveShot() method can take any Animal you give it. As long as the object you pass in as the argument is a subclass of Animal, it will work.



Dog's makeNoise() runs



Hippo's makeNoise() runs



A language like Java that allows objects to be of more than one type is said to support **polymorphic types**.